Chapitre 8

Notion de types paramétrés et Collections

1 Type paramétré

```
17 | IrreductibleGaulois copyPanoramix = (IrreductibleGaulois) ois.readObject() ;
```

Dans le chapitre précédent, on a utilisé l'instruction ci-dessus : à partir d'un `ObjectInputStream`, on a récupéré un objet. Dans l'exemple, on savait qu'on allait récupérer un objet de type `IrreductibleGaulois`, mais au moment de la compilation, on ne peut pas toujours vérifier ces transformations (pour rappel, on appelle cette transformation un transtypage explicite, « cast » en anglais). Ce n'est qu'au moment de l'exécution que l'on va détecter une erreur. Pour renforcer la sécurité, JAVA offre la possibilité d'utiliser un type paramétré.

1.1 Exemple

Commençons par un exemple dans lequel on nous demande de coder une structure de données très simple : la liste chaînée. Pour commencer, on va faire une liste chaînée de `String`. On veut donc faire une liste dans laquelle chaque `String` est encapsulé dans un noeud. Le noeud contient la valeur et un lien sur le noeud suivant. Pour se faire, on écrit une classe `Noeud` et une classe `ListeChainee` comme suit

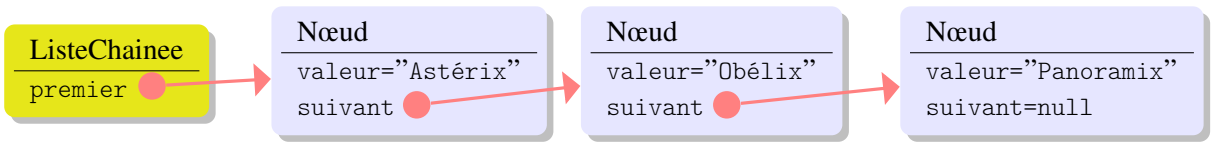
```
1  public class Noeud {
2     String valeur ;
3     Noeud suivant ;
4
5     public Noeud(String val){
6       valeur = val ;
7     }
8
9     public void setSuivant(Noeud next){
10      suivant = next ;
11    }
12 }
```

```java
public class ListChainee {
   Noeud premier ;

   public ListeChainee(){
     premier = null ;
   }

   public void add(String val){
     Noeud nouveau = new Noeud(val) ;
     if (premier == null)
       premier = nouveau ;
     else {
       Noeud dernier = premier ;
       while(dernier.suivant != null)
         dernier = dernier.suivant ;
       dernier.suivant = nouveau ;
     }
   }
}
```

Pour implémenter une liste contenant trois chaînes de caractères, par exemple {"Astérix","Obélix","Panoramix"}, en mémoire, on aura quatre objets que l'on peut représenter comme suit :



Maintenant, on voudrait faire une liste de `Personnages` à la place d'une liste de `String`.

— Une possibilité est de faire une nouvelle classe `NoeudPersonnage` dont les valeurs sont de type `Personnage` et faire de même pour une classe `ListeChaineePersonnages`. On devrait donc avoir une classe `NoeudType` par chaque type dont on voudrait faire une liste... pas très pratique. Imaginez que l'on veuille ensuite modifier la classe `Noeud` (par exemple ajouter une méthode, ou bien implémenter une liste avec une référence sur le noeud précédent), et il faudra faire le changement sur toutes les versions...
— Une autre possibilité est de modifier la classe `Noeud` et mettre `Object` à la place de String. On pourra mettre tous les types possibles dans un nœud, ce qui fonctionnera bien. Cependant, il faudra faire des transtypages explicites pour récupérer la valeur du nœud. Il faudra aussi faire bien attention à ce que l'on met dans la liste (on pourrait alors facilement mélanger des `Strings`, des `Personnages`, des `Integers`, etc... dans une même liste et il faudra un peu de travail utiliser son contenu par la suite).

La solution offerte par JAVA est la possibilité d'ajouter un type en paramètre de la classe. Ainsi, on va pouvoir avoir une seule implémentation de la classe `Noeud` qui va prendre en paramètre une classe, que l'on va noter `E` et on notera `Noeud<E>`. On va écrire les deux classes comme suit.

```java
public class Noeud<E> {
   E valeur ;
   Noeud<E> suivant ;

   public Noeud(E val){
     valeur = val ;
   }

   public void setSuivant(Noeud<E> next){
     suivant = next ;
   }
}
```

```java
public class ListChainee<E> {
   Noeud<E> premier ;

   public ListeChainee(){
     premier = null ;
   }

   public void add(E val){
     Noeud<E> nouveau = new Noeud<E>(val) ;
     if (premier == null)
       premier = nouveau ;
     else {
       Noeud<E> dernier = premier ;
       while(dernier.suivant != null)
         dernier = dernier.suivant ;
       dernier.suivant = nouveau ;
     }
   }

   public E get(int index){
     int i=0 ;
     Noeud<E> courant=premier ;
     while(courant.suivant != nll && i<index){
       i++ ;
       courant = courant.suivant ;
     }
     if(index == i) // on a trouvé l'élément numéro i
       return courant ;
     else
       return null ;
     }
   }
}
```

Avec cette implémentation, on va pouvoir utiliser des listes de `Personnages`, des listes de `String`, `Integer`, etc. Pour faire une liste de trois `Personnages`, on peut écrire le code suivant :

```
IrreductibleGaulois asterix = new IrreductibleGaulois("Astérix") ;
IrreductibleGaulois obelix = new IrreductibleGaulois("Obélix") ;
Gaulois Informatix = new Gaulois("Informatix") ;
ListeChainee<Gaulois> liste = new ListeChainee<Gaulois>() ;
liste.add(asterix) ;
liste.add(obelix) ;
liste.add(informatix) ;
```

Même si Astérix et Obélix sont des irréductibles gaulois, ils sont donc a fortiori des gaulois, et on peut donc manipuler une liste de gaulois. Maintenant, puisqu'il n'y a pas besoin de transtypage explicite, le compilateur peut vérifier si les types sont correctement utilisés.

Si on veut manipuler une liste de `Integer`, il faut ajouter des objets de type `Integer`, et quand on utilise un élément de la liste, on récupère un objet de type `Integer` et on doit utiliser une méthode comme `intValue()` pour obtenir la valeur de l'`Integer`. A priori, cela n'est pas très pratique, mais JAVA offre une fonctionalité qui fait automatiquement les conversions (ce qui est appelé *autoboxing*).

```
ListeChainee<Integer> maListe = new ListeChainee<Integer>() ;
//old style
maListe.add(new Integer(7)) ;
Integer sept = maListe.get(1) ;
System.out.println(sept.intValue()) ;
//new style
maListe.add(6) ;
int six = maListe.get(2) ;
```

1.2 Types paramétrés

Maintenant que nous avons présenté l'idée de base, nous allons donner plus de détails. On peut déclarer une classe avec plusieurs paramètres, la déclaration suit le modèle suivant :

```
class <nom classe> < paramètre 1 [, paramètre 2] >{
   //on peut déclarer des attributs des classes paramètres
  <paramètre 1> attribut ;
  //on peut déclarer par exemple des méthodes qui retournent l'attribut
  <paramètre 1> <nom méthode>(<liste arguments>){ ... }
   ...
}
```

L'héritage peut maintenant se faire à deux niveaux : soit au niveau de la classe elle même, soit au niveau des paramètres.

```
class <nom classe> < paramètre 1 [, paramètre 2] >
      extends <classe mère> < paramètre 1 [, paramètre 2] > { ... }
```

```
class <nom classe> < paramètre 1 extends <classe mère>[, paramètre 2] >
```

Il faut cependant faire attention à manipuler correctement l'héritage. Prenons un exemple :

```
1 ListeChainee<Gaulois> lg = new ListeChainee<Gaulois>() ;
2 ListeChainee<Personnage> lp = lg ;
```

La première ligne ne pose pas de problèmes particuliers. Sur la ligne 2, on a envie de dire qu'une liste de `Gaulois` est une liste de `Personnages`. Cependant...

```
3 lp.add(new Personnage("Jules César")) ;
4 Gaulois g = lg.get(1) ;
```

Sur la ligne 3, on ajoute un autre `Personnage` à la liste `lp` ce qui ne devrait pas poser de problèmes car `lp` est une liste de `Personnages`. Mais quand on récupère un élément via la liste `lg`, on ne récupère pas un `Gaulois` ! En fait, le compilateur JAVA *ne* permettra *pas* la ligne 2 : on ne peut pas faire passer une liste de gaulois comme une liste de personnages, sinon, on pourra mettre n'importe quoi dedans ! On peut donc se rappeler que si `F` est une classe de la descendance de la classe `M`, et si `G` est une classe générique, `G<F>` n'est pas dans la descendance de `G<M>`.

1.3 Jockers

Pour offrir plus de flexibilité, JAVA offre la possibilité d'utiliser des « jockers »qui va servir à exprimer un type *inconnu*. Par exemple, on va pouvoir écrire quelque chose comme
`ListeChainee<?> list = new ListeChainee<Gaulois>()` ; Dans cet exemple, on a une liste chainée d'inconnus appelée `list`. On ne pourra pas utiliser la méthode `add` car on devrait passer un élément de type `?`, mais comme on ne sait pas ce que c'est, cela pose problème. Par contre, on peut appeler une méthode comme `get` et dans ce car là, on peut au mieux faire une cast avec le type `Object`.

Ceci ne semble pas encore très intéressant, mais on va ajouter un élément important : on peut ajouter une *borne* au type inconnu. Par exemple, on peut écrire `ListChainee<? extends Gaulois>` pour dire que le type inconnu doit être dans la descendance de la classe `Gaulois`. Examinons l'exemple suivant pour comprendre ce à quoi peut servir cette possibilité. On voudrait écrire une méthode qui demande à tous les `Personnages` d'une liste de se présenter. Donc on voudrait écrire quelque chose comme :

```
1 public void presentezVous(ListeChainee<Personnage> liste){
```

Si on a une liste chaînée de gaulois, `ListeChainee<Gaulois> villageois`, on voudrait faire un appel à `presentezVous(villageois)`. Cela reviendrait à utiliser une liste de personnages pour manipuler une liste de gaulois... mais on vient de dire que cela est dangereux, et donc proscrit par JAVA . Donc, utiliser une liste de type `ListeChainee<Personnage>` comme paramètre de `presentezVous` n'est pas une bonne idée. On voudrait pouvoir avoir une liste de n'importe quelle sorte de `Personnages`. Examinons la déclaration suivante :

```
1 public void presentezVous(ListeChainee<? extends Personnage> liste){
```

Maintenant, la liste chaînée contient n'importe quel type qui hérite de `Personnage`. On peut donc passer une liste de `Personnages`, une liste de `Gaulois`, etc... Cependant, comme pour le cas précédent, JAVA ne nous permettra pas d'ajouter d'éléments dans la liste, on pourra seulement avoir accès aux éléments existents d'une liste.

1.4 Méthodes génériques

Il est aussi possible de définir des méthodes qui utilisent des types génériques. On prends ici la classe `List` qui fait partie de la bibliothèque de JAVA (cette classe se trouve dans `java.util`) et on en parlera dans la section suivante (Section 2).

```
1 public interface List<E> extends Collection<E> {
2    public <T> T[] toArray(T[] a) { ...}
```

La méthode `toArray` prend un paramètre `T` supplémentaire, ce qui permet à la méthode de prende n'importe quel tableau d'un type inconnu appelé `T` et de retourner un tableau de `T`.

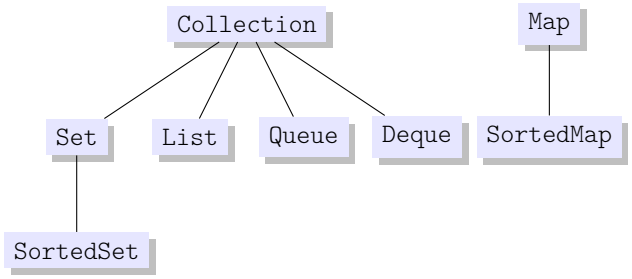
Dans la classe `Collections`, on trouve même des méthodes génériques avec des jockers. La méthode suivante par exemple sert à copier une liste dans une autre. Notez que le jocker ici à une borne inférieur, le type inconnu doit être un parent du paramètre T.

```
1 public class Collections {
2    public static <T> void copy(List<? super T> dest, List<? extends T> src ){ ...}
```

2 Collections

Les listes, les ensembles, les piles, les files d'attente sont des objets qui regroupent plusieurs éléments en une seule entité. Ces structures partagent un certain nombre de choses. On peut poser le même genre de questions : est-ce que la structure contient des éléments ? combien ? Certaines opérations sont similaires : on peut ajouter ou enlever un élément à la structure, on peut vider la structure. On peut aussi parcourir les éléments contenus dans la structure. On peut avoir des implémentations différentes de chacune de ces structures, par exemple on a vu l'exemple de la liste chaînée, mais on pourrait avoir une implémentation basée sur un tableau, ou un liste doublement chainée (avec une référence sur l'élément précédent).

Comment peut-on manipuler toutes ces structures ? Une solution – qui devrait paraître naturelle maintenant – est d'utiliser une hiérarchie d'interfaces, et c'est ce que JAVA propose.



— `Collection` : Tout en haut de la hiérarchie se trouve l'interface `Collection`. C'est le plus petit dénominateur commun, une collection rassemble simplement un nombre d'éléments. Certains types de collections autoriseront des doublons, pas d'autres, certains types sont ordonnés. On retrouve dans cette interface des méthodes de base pour parcourir, ajouter, enlever des éléments.
— `Set` : cette interface représente un *ensemble* au sens mathématique, et donc, ce type de collection n'admet *aucun* doublon.
— `List` : cette interface représente une *séquence* d'éléments. Ainsi, l'ordre dans lequel on ajoute ou on enlève des éléments est important. On peut avoir des doublons dans la séquence.
— `Queue` : cette interface représente une *file d'attente*. Dans une file d'attente, l'élément qui est en tête est particulièrement important. En fait si important qu'on peut avoir l'image suivante d'une file d'attente : il y l'élément en tête et il y a les éléments qui suivent, dont on ne préocuppe pas. On a généralement deux types de file d'attente : celle ou le premier entré est en tête de file et celui ou le dernier entré est celui en tête de file. L'ordre dans lequel les éléments sont ajoutés ou enlevés est important et il peut y avoir des doublons.

— `Deque` : cette interface ressemble aux files d'attente, mais les éléments importants sont les éléments en tête et en queue.
— `Map` : cette interface représente une relation binaire (surjective) : chaque élément est associé à une clé et chaque clé est unique (mais on peut avoir des doublons pour les éléments).
— `SortedSet` est la version ordonnée d'un ensemble
— `SortedMap` est la version ordonnée d'une relation binaire ou les *clés* sont ordonnées.

On introduira de la notion d'ordre entre des objets plus tard dans ce document, mais elle sera vu pendant le cours *Programmation Java Avancée*.

Notez que ces interfaces sont génériques, i.e. on peut leur donner un paramètre pour indiquer qu'on a une collection de `Gaulois`, de `Integer`, de `String`, etc...

2.1 Méthodes de l'interface Collection

Toutes les classes qui implémentent l'interface `Collection` devront redéfinir un certain nombre de méthodes parmis lesquelles on retrouve des méthodes pour l'ajout, le retrait d'éléments, des méthodes pour savoir si un ou des éléments sont présents dans la collection, une méthode (`size`) pour connaitre le nombre d'éléments contenus dans la collection. On ne va pas expliquer en détail chaque méthode, mais vous trouverez ci-dessous une liste (non exhaustive) des méthodes de cette interface. Notez la présence de certaines méthodes génériques. On va expliquer dans la section suivante l'utilité de la méthode `iterator()`.

— `boolean add(E e)`
— `void clear()`
— `boolean contains(Object o)`
— `boolean equals(Object o)`
— `boolean isEmpty()`
— `Iterator<E> iterator()`
— `boolean remove(Object o)`
— `int size()`
— `Object[] toArray()`

2.2 Parcourir une collection

Il y a deux moyens de parcourir une collection : soit en utilisant une boucle « pour chaque élément », qui offre une version différente de la boucle `for` vue précédemment, soit en utilisant un objet appelé `iterator`.

En utilisant la généricité, le compilateur va connaître le type des éléments qui sont contenus dans la collection. Une collection étant un ensemble d'éléments, JAVA propose une façon simple d'accéder à chacun des éléments. Dans l'exemple ci-dessous, on a une collection `maCollection` qui contient des objets de type `E`. On va accéder à chaque élément de la collection `maCollection` en utilisant le mot-clé `for`, chaque élément sera stocké dans une variable `<nom>` de type `E` (évidemment).

```
1 Collection<E> maCollection ;
2 ...
3 for (E <nom> : maCollection)
4     // block d'instructions
```

Par exemple, dans l'exemple suivant, on parcours une liste de `Personnages` et on appelle la méthode `presentation()`.

```
1 LinkedList<Personnage> villageois = new LinkedList<Personnage>() ;
2 villageois.add(new Gaulois("Ordralfabétix")) ;
3 villageois.add(new Gaulois("Abraracourcix")) ;
4 villageois.add(new Gaulois("Assurancetourix")) ;
5 villageois.add(new Gaulois("Cétautomatix")) ;
6 for (Personnage p : villageois)
7     p.presentation() ;
8
```

Une autre solution est d'utiliser un objet dédié au parcourt d'éléments dans une collection : un objet qui implémente l'interface `Iterator`. Pour obtenir cet objet, on peut appeler la méthode `iterator()` qui se trouve dans l'interface `Collection`. L'interface `Iterator`, contient les deux méthodes suivantes :

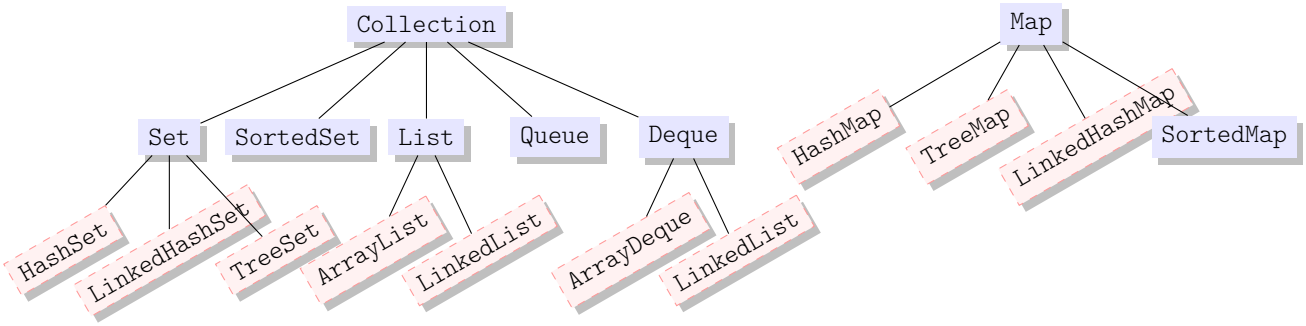
— `hasNext()` retourne un `boolean` qui indique s'il reste des éléments à visiter,
— `next()` donne accès à l'élément suivant, et `remove` permet d'enlever l'élément de la collection.

```
1 public interface Iterator<E> {
2    boolean hasNext() ;
3    E next() ;
4    void remove() ; //optional
5 }
```

En fait, l'utilisation d'une boucle « pour chaque »masque l'utilisation d'un `Iterator`. En général, l'utilisation de la boucle « pour chaque »est la plus simple. Vous devez utiliser un `Iterator` lorsque vous voulez enlever un élément ou lorsque vous voulez parcourir plusieurs collections en parallèle.

2.3 Implémentations

Pour chacune des interfaces, il existe plusieurs *implémentations*. Le diagramme ci-dessous indique les principales implémentations. Certaines sont basées sur des tableaux pour stocker les éléments (`ArrayList`, `ArrayDeque`), d'autres sur des arbres (`TreeSet`, `TreeMap`), d'autres avec des tables de Hash (`HashSet`, `HashMap`). Chaque implémentation a ses propres propriétés (avec ses propres avantages et défauts). Mieux vaut donc lire la documentation pour savoir quelle implémentation est plus appropriée à chaque utilisation. Pour des utilisations « simples » (où les collections ne contiennent pas un trop grand nombre d'objets) on ne verra guère de différences entre les implémentations. Nous ne prendrons pas le temps ici de comparer toutes ces implémentations.



2.4 Ordre

La *classe* `Collections` (à ne pas confondre avec l'interface `Collection`) est une classe qui contient beaucoup de méthodes statiques pour manipuler des collections passées en paramètres. Par exemple, il existe une méthode `sort(List<T> maListe)` qui va trier une liste `maListe`. Si la liste contient des dates, la méthode va trier en ordre chronologique, si la liste contient des `String`, la liste sera trier par ordre lexicograhique. Mais si vous voulez trier des `Gaulois` par quantité de sangliers consommés par an, vous pourrez aussi facilement utiliser la méthode `sort` ! Derrière tout cela, une nouvelle interface est utilisée : l'interface `Comparable`.

Cette interface ne contient qu'une seule méthode : la méthode `public int compareTo(T o)`. Cette méthode retourne un entier négatif si l'objet est plus petit que l'objet passé en paramètre, zéro s'ils sont égaux, et un entier positif si l'objet est plus grand que l'objet passé en paramètre.

Si vous regardez la documentation des classes `String`, `Integer`, `Double`, `Date`, `GregorianCalendar` et beaucoup d'autres, vous verrez que ces classes implémentent toutes l'interface `Comparable`. Ils ont donc une implémentation de la méthode `compareTo`.

Vous pouvez donc spécifier votre propre méthode `compareTo`. Par exemple pour la classe `Gaulois`, on pourrait écrire

```
1 public class Gaulois extends Personnage implements Comparable<Gaulois>{
2    String nom ;
3    int quantiteSanglier ;
4      ...
5
6    public int compareTo(Gaulois ixis) {
7      return this.quantiteSanglier - ixis. quantiteSanglier ;
8    }
9 }
```

Attention, si vous essayez de trier une liste qui n'implémente pas l'interface `Comparable`, vous provoquerez une erreur (`ClassCastException`).